CSC2002S PCP Assignment 1

Jing Yeh
YHXJIN001

Table of Contents

1. Introduction and Methods

1.1 General Introduction

This assignment centres around parallel computing. We were provided with the serial program that simulates Abelian sandpiles. Our task is to code a parallel version of the same program, and then report on the potential speed ups for different grid sizes.

I have included make commands to run both the serial and parallel versions, as well as a few commands to run some python scripts to help with the experiment. A summary is given below.

To run a single of the serial/parallel version of the program

```
make run_serial $(RUN_ARGS="input_path output_path")$
make run_parallel $(RUN_ARGS="input_path output_path")$
```

To run a series of trials of both the serial and parallel version of the programs **Note:** results are automatically written to *"analyses/resultsParallel.csv"* and *"analyses/resultsSerial.csv"* respectively.

```
make run_test $(TEST_INFO="start end intervalSize(>=1) values")$
// For example:
make run_test TEST_INFO="500 1000 100 6"
// Will run 5 runs of both the serial and parallel versions // of the
program on 500x500 600x600 ...... 1000x1000 grids // with each cell
filled with 6
```

A csv generator to quickly create grids filled by the same value is included for convenience

```
make csv $(CSV_INFO="rows columns value")$
// For example:
make csv CSV_INFO="517 517 8"
// Will genearte a csv file with 517 rows, 517 columns, with // each
cell filled by 8
```

1.2 Parallelisation Approach

The parallelisation was carried out as followed:

- Calculate the cutoff value using the empirically determined formula

  $$\frac{2400}{processors}$$

  Note: The justification for the formula is that the parallel algorithm tend to start surpassing the speed of the serial algorithm when the grid size is greater than 300-500. This formula preserves the inverse relationship between cutoff and the number of processors, but is also adjusted to not be using too many threads for smaller grid. I arrived at the value 2400 through trial and error.

- We will then recursively divide the grid into smaller sub-grids, using the *compute* method of a wrapper class *ParalleliseGrid*. The wrapper class is extending from *RecursiveTask*, and parallelise the *Grid* object
- Once the sub-grid size goes below the cut-off value. Weupdate the sub-grid by saving the updated value to a local grid, the *localUpdatedGrid*, each stored by an instance of *ParalleliseGrid*.

- After the updates are done, I then merge the local grids. The merged grid will then be passed to the *nextTimeStep()* method of the *Grid* class to copy the values in the *mergedGrid* to the actual grid
- This process will repeat indefinitely until there are no more changes.

1.3 Validation of Algorithm

A quick way to check for the correctness of the parallel algorithm is to simply try spotting any differences between the images generated by the serial and the parallel versions of the simulation. One can also check if the number of steps perform by both algorithms are the same. For more accurate examination, however, more robust methods are needed.

I have therefore included a method in both versions of the programs to output the final grid as a csv file. The csv files generated could then be compared by the *validate_results.py* script I have included. The script uses the *compare()* method that comes with the widely used *Pandas* library

The script could be run with the following command

```
make validate_results VALIDATE_ARGS="csv1_path csv2_path outputTxt_path"
```

Note: The user might need to pip install Pandas first before running the script

1.4 Benchmarking

I tested the algorithms on a range of grid sizes: 100x100, 200x200,...1000x1000, with each cell filled by 4, using the run_test.py to replicate the tests 10 times, on a 4-core machine (Windows Surface Laptop 2, Intel i5-8250U) and another 8-core machine (Macbook Air 2020, Apple M1). I then compare the speed of of the two algorithms for a given grid size with the following formula:

$$speedUp = \frac{ParallelTime}{SerialTime}$$

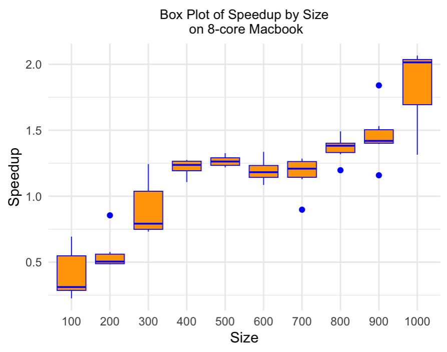
the speed up results for each grid size are then used to produce a box plot and a connected scattered plot using the mean speed up value for each grid size.

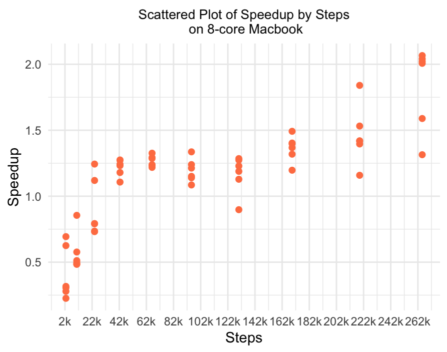
Here's the command that was run for 10 times:

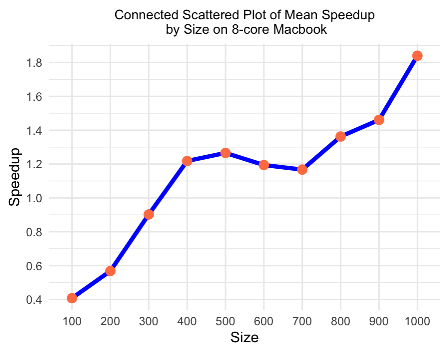
```
make run_test TEST_INFO="100 1000 100 4"
```

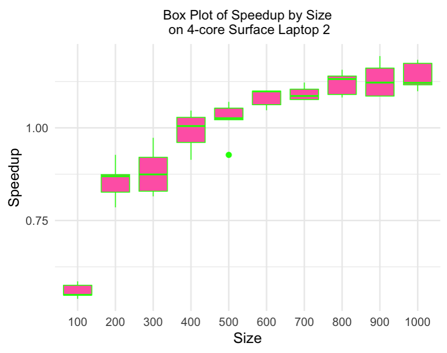
2. Discussion

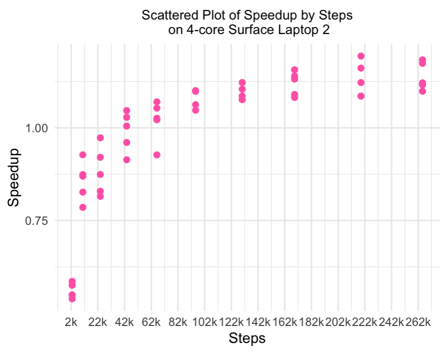
2.1 Speedup Graphs

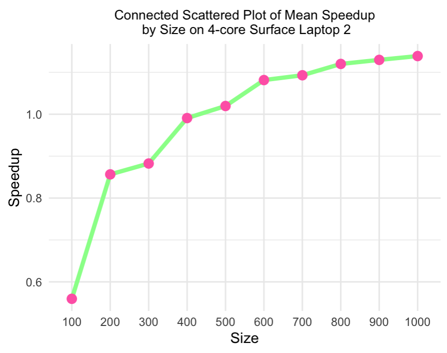












2.2 Effects of Different Data Sizes

For smaller grids (grid size < 300x300 for 8-core and 500x500 for 4-core), the serial algorithm tend to run faster than the parallel algorithm. This is due to the overhead caused by multithreading. However, as the grid size increases, we can observe an increase in the speed up value as well. This is where parallel programming shines, as the benefit of dividing tasks amongst threads begins to outweighs the effects of overhead. This is known as soft scaling.

2.3 Effects of Different Computer Architecture

Theoretically, the ideal speed up for the 8-core machine should be 2x greater than that of the 4-core machine - this is generally the case. However, thanks to the hyperthreading technology that extends, achieved partly through time-slicing, and I have no way of controlling which threads the program would run on, the speedup I observed is not as significant.

2.4 General Trends

As discussed, the speed of the parallel algorithm begins to surpass the serial version as the grid size increases. Further, I can also observe increase in speed as the number of cores of the machine increases.

There is a spike in speedup for grid size of 400 because this is the value that the cutoff value begins dividing tasks amongst multiple threads. Sometimes the parallel algorithm can suddenly take longer than usual to run. This is mostly because too many applications are being run in the background.

3. Conclusion

I have run the parallelisation on two different machines: one 8-physical -ore machine (Apple M1) with 1 thread per core, and a 4-physical core machine (intel i5-8250U) with 8 logical cores (2 threads per core). I found that, for the relatively small data sets tested, significant speed up can be observed on the Macbook as the size of the grid increases, and a minor increase in speed up for the Windows Surface Laptop 2. Overall, parallelisation is worthwhile as long as the data size / number of steps required to complete the task is relatively large and CPUs with multiple physical cores are available.